Automated Testing of Planning Models

Klaus Havelund, Alex Groce, Gerard Holzmann,
Rajeev Joshi, Margaret Smith

Jet Propulsion Laboratory*, California Institute of Technology
4800 Oak Grove Drive, Pasadena/Los Angeles, CA 91109
{klaus.havelund,alex.d.groce,gh,rajeev.joshi,margaret}@jpl.nasa.gov

Abstract – Automated planning systems (APS) are maturing to the point that they have been used in experimental mode on both the NASA Earth Orbiter 1 satellite and the Deep Space 1 spacecraft. One challenge is to improve the test coverage of APS to ensure that no unsafe plans can be generated. Unsafe plans can cause wasted resources or damage to hardware. Model checkers can be used to increase test coverage for large complex distributed systems and to prove the absence of certain types of errors. In this work we have built a generalized tool to convert the input models of an APS to PROMELA, the modeling language of the SPIN model checker. We demonstrate on a mission sized APS input model, that we can explore the space of possible plans in PROMELA and use SPIN to verify with high probability the absence of unsafe plans.

1 Introduction

Automated Planning Systems (APS) have performed onboard planning and commanding in experimental mode for two NASA technology validation missions: DS1 and EO1. APS are also used to support ground planning of sequences for both the Mars Exploration Rover and the Phoenix missions. Unlike traditional software, which executes a fixed sequence, an APS takes a few high level goals, and an input model and automatically generates a sequence of actions, called a *plan*, that achieves the goals. An APS can respond to unexpected situations and opportunities that a fixed sequence can not. The same flexibility that makes it possible to respond to unanticipated situations also makes a planner far more difficult to verify. If a mission manager is to trust an APS to autonomously command, it must be shown to generate the correct plan for a vast number of situations. Empirical test cases can cover only a handful of the most likely

* The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

or critical situations. Formal methods can in principle prove that every plan meets certain properties and can prove the absence of a dangerous or undesirable plan.

In this work, we expand upon the results of our previous work [1] that demonstrated that it was possible to apply formal methods, and in particular, the Spin model checker [2–4] to improve test completeness when verifying APS input models. In particular, we have constructed a tool called Map to automate the conversion APS models to Promela, the language of the Spin model checker. We have demonstrated that a large portion of the semantics of an APS model is expressible in the language of the model checker. As the subject of this work, we selected the Aspen APS and its modeling language Aml [10, 12–14] developed by Jet Propulsion Laboratory (JPL) because it is currently successfully commanding the Earth Observer 1 (EO1) Autonomous Sciencecraft Experiment onboard the EO1 satellite.

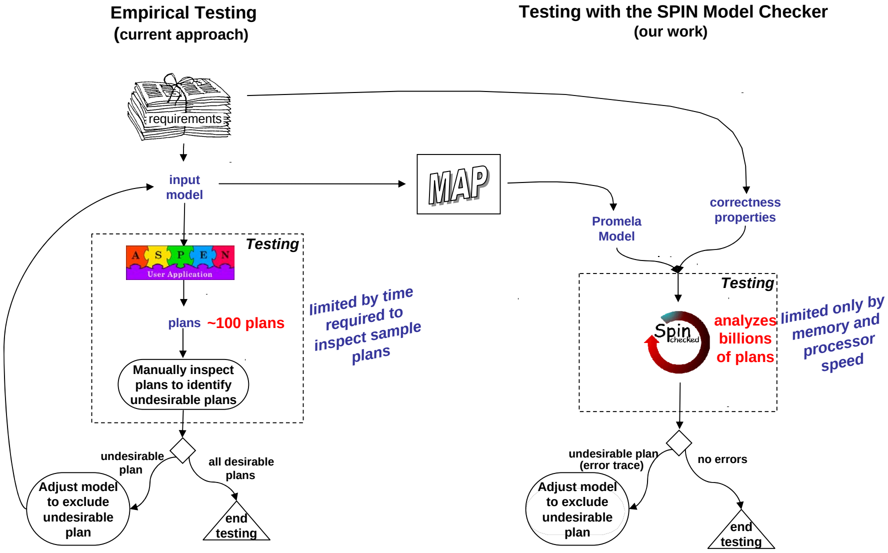

Empirical Testing
(current approach)
Testing with the SPIN Model Checker
(our work)
requirements
input model
MAP
Promela Model
correctness properties
Testing
ASPEN
User Application
plans
~100 plans
Manually inspect plans to identify undesirable plans
limited by time required to inspect sample plans
Testing
Spin
analyzes billions of plans
limited only by memory and processor speed
undesirable plan
all desirable plans
Adjust model to exclude undesirable plan
end testing
undesirable plan (error trace)
no errors
Adjust model to exclude undesirable plan
end testing

Fig. 1. Map in context.

The traditional approach to testing a plan model is to use ASPEN to exercise the model with various goals, and manually examine the generated plans, see Figure 1. The MAP conversion tool offers an alternative approach where an AML model is translated to a PROMELA model, such that the SPIN model checker can be used to test the plan model. The tool handles goals, activity decomposition, temporal constraints, and automated calculation of a cone of influence to reduce the search space. We demonstrate that the substantial increase in test coverage achieved through the use of model checking can work in practice and scale to a mission sized AML input model.

In work that predated publication of our previous paper [1], the real-time model checker UPPAAL was used to check for violations of mutual exclusion properties and to check for the existence of a plan meeting a set of goals [5]. In contrast, the work reported in this paper shows that for verification of a set of properties of interest, it is not necessarily required to reason about time. SPIN has also been used to verify plan execution engines [6, 7]. Automatically generated test oracles have been used to assist in the interpretation of test plan outputs from APS [8]. A comparison of three popular model checkers, SPIN, SMV and Murphi showed that these model checkers can be used to check for the existence of a plan meeting a set of goals [9].

The rest of the paper is organized as follows. Section 2 briefly describes the ASPEN planner and the SPIN model checker. Section 3 presents an example of an AML model, and how SPIN is used to explore the PROMELA model generated by the MAP tool. Section 4 explains the principles of the translation from AML to PROMELA. Section 5 presents the results of analyzing the EO1 model. Finally, Section 6 concludes the paper and suggests future work.

2 The ASPEN Planner and the SPIN Model Checker

2.1 The ASPEN Planner

The ASPEN planner takes as input: an initial state, a goal, and a plan model describing allowable activities and constraints on their relationships; and produces a plan of activities that achieves the goal while satisfying the constraints in the model. In order to be efficient for on-board planning, the ASPEN planner performs a heuristics-based search, not exploring all possible paths, but instead only exploring a minimal search space. The objective of the planner is to find a single good plan, and the assumption is that such a plan exists. While this minimal search approach makes Aspen efficient for finding plans quickly when they exist, it makes Aspen's search incomplete, which is a drawback during testing. For instance, if Aspen does not return a plan, one cannot conclude that there is no plan.

An Aml model consists of a set of goals, activity specifications, resources, and states. Arbitrary C or C++ functions may be called from the model to calculate values used to determine resource requirements and states. The start of an activity is normally guarded so that the activity can only be scheduled if necessary resources are available and if the spacecraft is in a desired state. Activities typically modify states and resources at the beginning and/or end of the activity. Activities can be decomposed recursively into lower level, sub-activities. A number of temporal relations can be defined to order the start and completion of sub-activities with respect to one another. States and resources are used in Aml models to constrain the types of plans that are generated to a set that will be safe and feasible. For instance, an atomic resource such as a solid state recorder (SSR), that can only be safely accessed by one reader or writer at a given time, will be tracked by a mutex state. An activity that needs to write to the SSR will have a guard that prevents the activity from starting until the SSR lock is available. The activity needing to read or write to the SSR takes the lock upon entry and restores it upon exit.

A tightly constrained Aml input model will have a smaller number of potential plans, and can be more completely tested, but will be less agile in responding to unexpected events during spacecraft operation. A less tightly constrained model exploits the strengths of the APS system to respond to the unexpected, but in order to be trusted, must be more thoroughly tested than is possible with standard test techniques.

2.2 The Spin Model Checker

Spin is a model checker and can analyze the correctness of finite state concurrent systems with respect to formally stated properties [3]. A particular concurrent system is formalized in PROcess MEta LAnguage (Promela), and correctness properties to be verified can be formalized in Linear Temporal Logic (Ltl), a visual tool such as the TimeEdit tool [11] that generates Buchi automata, or using assertions placed in the Promela model. The Spin tool also provides a simulator, with which Promela models may be executed. This can in particular be used to re-run error traces generated by the model checker for properties that are not satisfied. Spin's search attempts to be exhaustive, continuing until it finds an error, memory is exhausted, or the search completes. The correctness property can express a desired behavior, like a goal in Aspen's Aml language, or an undesired behavior, such as a unsafe plan that should be excluded from an Aml input model.

Promela is Spin's modeling language, supporting the declaration of process types, and instantiation (running/spawning/starting) of instances of these types. The language can be thought of as a multi-threaded programming language. Processes communicate via shared variables and/or by message passing through communication channels. A process can block by waiting for a Boolean predicate over the global variables to become true, or it can block on waiting for a value to appear on an input channel. The execution of a Promela model consists of executing these parallel running processes in a non-deterministic interleaved manner until no process can continue, either because all processes have terminated normally, or they have deadlocked. A Promela model denotes the set of all such finite and infinite execution traces. The Spin model checker conceptually explores all traces for conformance to or violation of a formal property.

3 Example

The following example is intentionally made as small as possible (and consequently rather artificial), but sufficiently complex to still illustrate the fundamental principles. The scenario is the operation of a planetary rover performing drilling activities on another planet. First an Aml model is represented. Second, it is shown how Spin is used to analyze the Promela model generated by Map. In this section the generated Promela will be regarded as a black-box, not unlike how a user would perceive it. In Section 4 the translation will be explained.

```
resource power {
   type = depletable;
   default_value = 75;
   capacity = 100;
   min_value = 10;
}

resource buffer { type = atomic; }

state_variable buffer_sv {
  states = ("empty","full");
  transitions = ("empty"->"full", "full" -> "empty");
  default_state = "empty";
};

activity drill {
  string hole;
  int depth;
  int power_use;
  dependencies = power_use <- powerof(depth);
  reservations =
    buffer,
    buffer_sv must_be "empty",
    buffer_sv change_to "full" at_end,
    power use power_use;
}

activity uplink {
  reservations =
    buffer,
    power use 30;
}

activity charge {
  reservations = power use -25;
}

activity experiment {
  decompositions =
    (drill with ("hole1" -> hole, 7 -> depth),uplink,charge
      where charge ends_before end of drill)
    or
    charge;
}
```

Fig. 2. AML model of drilling scenario.

3.1 Aml Model of Drilling Rover

The rover can perform three activities: (i) *Drill*: the rover drills a hole of a certain depth, extracts some soil, and performs some analysis on the selected material, for example using an oven. All these activities are here abstracted into the single drill action. (ii) *Uplink* : when the drilling (and included analysis) has been performed the results must be uplinked to a spacecraft (which subsequently transmits it to earth, not modeled). (iii) *Charge* : the drilling as well as the uplink both require power, represented by a power resource. This resource can be charged with new energy when becoming low. The Aml model presented in Figure 2 formalizes this scenario. Our goal will be to generate plans that request drilling and uplink of the results, with charging occurring as needed. We shall illustrate how Map can be used to detect various errors in the model to be presented.

The rover and the equipment on board the rover uses various resources. There are two types of resources: atomic, and variable. Atomic resources are physical devices that can only be used (reserved) by one activity at a time (for example a science instrument). A variable resource has at any point in time a value and can be used by more than one activity at a time, each reducing the quantity of the resource, as long as the minimum/maximum bounds are not exceeded. A variable resource is either depletable or non-depletable. A depletable resource's capacity is diminished after use (for example a battery), in contrast to a non-depletable resource, where the used quantity is automatically returned (for example solar power). The model contains one variable depletable power resource (lines 01–06). The power resource has a current starting value of 75, a minimum value of 10 (it cannot go below) and an maximum capacity of 100. Digital results collected during drilling are stored in a data buffer before being uplinked. The data buffer is modeled as an atomic resource (line 08) and will be reserved by the drilling and the uplink activities to ensure mutual access. In addition, a state variable (lines 10–14) is introduced to model the status of the buffer: whether it is empty or full. The state machine has two states (empty and full) and two transitions: one from empty (the initial state) to full, and one from full back to empty.

The drilling activity (lines 16–26) declares three local variables hole, depth and power_use (lines 17–19). Any local variable in Aml can function as a parameter. The first two will function as parameters (what hole to drill and what depth), while the third is a real local variable holding how much power to consume, being assigned a value in a dependency clause (line 20) as a function of the depth. The drilling activity reserves a collection of resources (lines 21–25): the data buffer (line 22, ensuring mutual exclusion during use), which must be "empty" (line 23), and will transition to "full" after (line 24); and power as a function of the depth of the hole (line 25). The uplink activity (lines 28–32) reserves the buffer from where data are uplinked and uses 30 power units. The charge activity (lines 24–36) adds 25 units back to the power resource, (using AMLs semantics of providing negative numbers when adding, and positive numbers when subtracting).

The main activity is called experiment (lines 38–44) and is decomposed into the three activities: charge, drill and uplink. The decomposition consists of *either* (lines 40–41) performing a drill, an uplink and a charge, where the charge is required to end before the end of the drill (to save time), *or*, if there is not power enough, to just charge the rover with new energy (line 43). Note the constraint: '`charge ends_before end of drill`'. AML allows for several kinds of constraints , '*A constraint B*', between two activities A and B (that can occur in any order if no constraints are given): `contains`, `contained_by`, `starts_before`, `ends_before`, `starts_after`, `ends_after`, all further followed by one of `start of`, `end of`, or `all of`. Examples are: `A starts_before start of B`, `A starts_after end of B`, and `A contains all of B` (the B activity occurs during the A activity, not before and not after).

An initialization file outlines what activities should be instantiated. In this case one instance of the experiment activity is initiated:

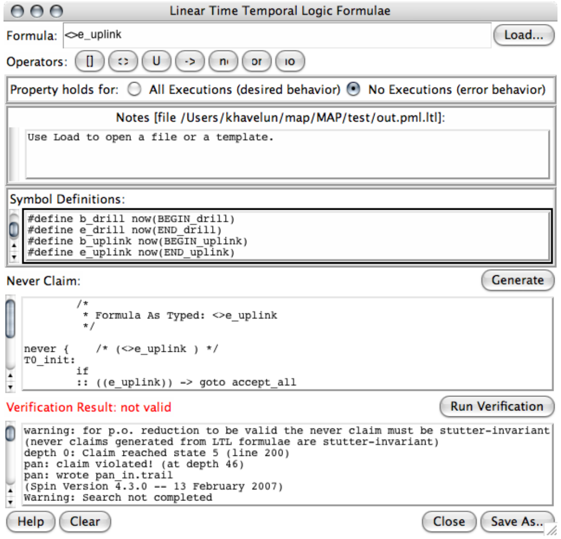
```
experiment exp {}
```

Note that the experiment activity itself launches the charge, drill and uplink activities through decomposition.

3.2 Analyzing the Model with SPIN

Verification 1 In order to verify LTL properties with SPIN, atomic conditions (PROMELA macros using `#define`) have to be defined. For example, the event `e_uplink` will become true when the uplink activity terminates. For each activity A, there will be a `b_A` (begin A) and a `e_A` (end A) event, which can be referred to in SPIN. The first property we will verify is that eventually an *end of uplink* is observed. This is achieved by asking SPIN to prove that there is no execution satisfying the LTL property `<>e_uplink` (see Figure 3).

The property states that eventually the end of an uplink occurs. A trace satisfying this property should constitute in a good plan. By making SPIN attempt to verify that an execution satisfying this property does *not*



Formula: <>e_uplink

Property holds for: ○ All Executions (desired behavior) ● No Executions (error behavior)

Notes [file /Users/khavelun/map/MAP/test/out.pml.ltl]:

```
Use Load to open a file or a template.
```

Symbol Definitions:

```
#define b_drill now(BEGIN_drill)
#define e_drill now(END_drill)
#define b_uplink now(BEGIN_uplink)
#define e_uplink now(END_uplink)
```

Never Claim:

```
        /*
         * Formula As Typed: <>e_uplink
         */

never {    /* (<>e_uplink ) */
T0_init:
        if
        :: ((e_uplink)) -> goto accept_all
```

Verification Result: not valid

```
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
depth 0: Claim reached state 5 (line 200)
pan: claim violated! (at depth 46)
pan: wrote pan_in.trail
(Spin Version 4.3.0 -- 13 February 2007)
Warning: Search not completed
```

Fig. 3. XSPIN – generate a plan ending in an uplink.

exist, we use SPIN to generate an error trace (a plan) that achieves such a state in case it exists. Note that we have chosen the *No Executions* option in XSPIN in order to get an error trace (plan). The verification causes XSPIN to generate the message sequence diagram shown in Figure 4.

The message sequence diagram shows for each activity (modeled as a PROMELA process, see Section 4) a vertical time line, showing when it begins and when it ends. In this case it is observed that there is an uplink before any drilling has taken place. This is an error according to our informal requirements. By studying the model it is detected that the uplink activity does not check the status of the data buffer to see whether it contains data before the uplink takes place. The buffer must be full before uplink (a check on the buffer state variable), and after the uplink it must be set to empty. To fix this we modify the Uplink activity as follows:

```
activity uplink {
    reservations =
        buffer,
        buffer_sv must_be "full",    // added
        buffer_sv change_to "empty", // added
        power use 30;
```

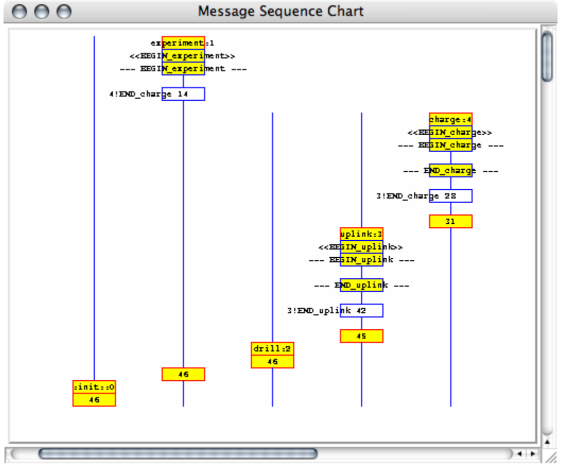


Fig. 4. XSPIN – an error trace equals a plan.

```
    }
```

3.3 Verification 2

Retrying the verification after this modification yields *no errors*. However, no errors means no plan. Recall that SPIN is asked to prove that there is no execution leading to an uplink. After further examination it is discovered that even though the charge activity adds 25 units, which should be enough to cover the combined usage of 70 (drill) plus 30 (uplink) with an initial resource value of 75, another 10 needs to be added since the minimal value of the resource is set to 10 (cannot go below). The maximum capacity must consequently also be increased. The power resource therefore needs to be modified as follows:

```
    resource power {
        type = depletable;
        default_value = 85; // changed from 75 to 85
        capacity = 110;     // changed from 100 to 110
        min_value = 10;
    }
```

This time an acceptable sequence of events is generated: first drilling, then a charge, and then uplink.

3.4 Verification 3

We have now demonstrated that there is a plan that ends in an uplink preceded with a drill. The question is whether there are any plans that end in an uplink without being preceded with an drill. We can verify this by searching for a plan satisfying the following LTL property: `!e_drill U e_uplink`. That is: no drill until an uplink. The until operator of LTL is strong, hence this means that an uplink must occur (and no drill before that). Since we want to show that there is no such plan, we enter this property with *No Executions* set. The verification shows that there are no such executions (errors : 0), which is a satisfactory result.

All our properties so far have been stated as the LTL property `<>goal`, using the *No Executions* option to make SPIN attempt finding just one execution that makes the goal true. It turns out that for verification of plan models this seems to be the most natural verification style: to postulate the non-existence of an execution (plan) that satisfies a particular property. It is, however, possible also to use the *All Executions* option in XSPIN. That is, to prove that for all execution traces some property is true. Note though that a plan model denotes executions that lead nowhere. Such blind alleys are simply part of the search problem. Hence, one has to be careful when stating properties to be true on all executions. One has to limit the verification to only those executions that achieve some meaningful goal. In our last case we can state the property that: every uplink is preceded by a drill as the following property to be true on all traces, knowing that there is only one uplink possible: `<>e_uplink -> <>(e_drill & <>e_uplink)`. That is, "for all traces, if the trace is a good plan (eventually from the beginning of the trace there is an uplink), then (also from the beginning of the trace) there is a drill, followed by a (the) uplink". This is, however, a slightly complicated way of stating our desired property.

4 Translation from AML to PROMELA

Planning in principle can be regarded as the following problem: given is a model $M = (\Sigma, A)$ consisting of a state Σ (resources and state machines), and a finite set of activities $A = A_1, A_2, A_3, \ldots, A_n$ that access variables in the state Σ. Each activity A_i has a precondition $pre - A_i$ on the state Σ that has to be true before that activity can execute (or *be put down on a time-line*, using planning terminology), and a post-condition $post - A_i$, defining a side-effect on the state Σ. The activities can be thought of as guarded commands. A planning problem is a triple (I, G, M) consisting of an initial state I and a goal state G to be achieved from the initial state while obeying the model M (obeying the pre-conditions essentially). The planning problem is obviously more complicated, in particular in the case of AML, which allows for dynamically created activities and time constraints.

However, this view of the planning problem directly leads to a process view of planning: given a set of processes (activities), find an execution of these that leads from the initial state to the goal state, without deadlocking or otherwise failing in between. This is the view underlying the MAP translator. It translates an AML model into a PROMELA model of concurrent processes, one for each activity, with a pre-condition and a post-condition. Concurrency is normally regarded as a hard problem for users to get right, and the above argumentation suggests that the planning problem is equally difficult to get right.

More specifically, an AML model is translated into a PROMELA model, which contains a process type (proctype) for each activity. The body of each such process type consists of two sequentially composed statements $S_1; S_2$: a beginning S_1 and an ending S_2, each of which is an atomic statement (encapsulated with PROMELAs atomic-construct). The basic idea is that the scheduling of an AML activity A over a time period starting at time t_1 and ending at time t_2 in SPIN will result in the corresponding process executing its first atomic statement S_1 at a point corresponding to time t_1 and its second atomic statement S_2 at a point corresponding to time t_2. However, since SPIN does not model real-time, time periods are not measured, only the relative ordering of events is modeled. Planning in SPIN consists of finding an execution trace that executes the processes (respecting the guards of course) in such a manner that a specific end state is reached, with the expected processes executing in a desired order, and such that the state satisfies some invariants during the execution.
Resources are declared as state variables that get written to and read from during the "execution" of the PROMELA model:

```
int power;
bool buffer;
byte buffer_sv;
int buffer_sv_reserve_count;
```

The power variable holds current power level. The buffer variable represents a semaphore, which is either taken (value 1) or free (value 0). The buffer state variable (buffer_sv) holds the current state of the buffer state machine. The buffer_sv_reserve_count is increased each time a process performs a `must_be` request, as for example the drill action in line 23. The drill action here requires the state variable to have this value throughout its execution. Several activities can require this to be true, and all be able to execute at the same time. Each process will count this variable up at entry and down on exit, and the state variable (buffer_sv) itself cannot change unless this counter is 0.

As already mentioned, an activity is modeled as a process. SPIN attempts to "execute" processes, thereby producing an execution trace, which becomes the sought plan. In the example, the experiment activity starts the three sub-activities drill, uplink, and charge, with the constraint that the charge should end before the end of the drill action. In addition, the three sub-activities should all terminate before the end of the experiment activity since they are created as sub-activities (AML semantics). These constraints are illustrated in Figure 5.

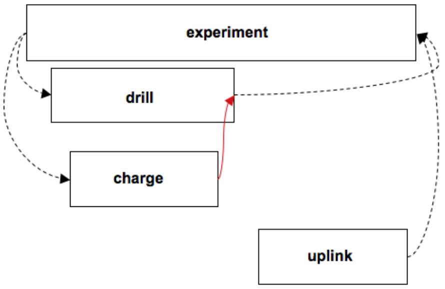


Fig. 5. Activity constraints. Stipled lines are constraints imposed by AML semantics. Fully drawn constraint comes from the model constraint: "`charge ends_before end of drill`".

These constraints are imposed in the Promela model by passing two sets (collections) of events to each process: those that it should wait for before it starts, and those it should wait for before it terminates. In the above case, for example, the drill process should be passed the sets: $\emptyset$ (don't wait to start) and $\{end\_charge\}$ (wait for charge to terminate before terminating). In order to know what events actually happened in the context (parent) in which a process exists, it takes a third parameter, a reference to a set that is continuously updated with events as they happen. The generated process declaration in Figure 6 contains these parameter definitions.

```
proctype drill(set begin_events; set end_events;
               set external_events; short sigstart; short sigend;
               int depth)
{
  byte _e_;
  int power_use;
  atomic {
    subset(begin_events,external_events);
    power_use = powerof(depth);
    (buffer==1 && buffer_sv==ENUM_empty &&
     (power-power_use)<=110 && (power-power_use)>=10 ) ->
    buffer = buffer-1;
    buffer_sv_reserve_count = buffer_sv_reserve_count+1;
    power = power-power_use;
    addorlog(external_events,sigstart)
  };
  atomic {
    subset(end_events,external_events);
    (buffer_sv==ENUM_empty &&
      (buffer_sv_reserve_count==1 || buffer_sv==ENUM_full)) ->
    buffer = buffer+1;
    buffer_sv_reserve_count = buffer_sv_reserve_count-1;
    buffer_sv = ENUM_full;
    addorlog(external_events,sigend)
  }
}
```

Fig. 6. Promela model of drill activity.

The first two parameters are the sets of events to wait for before starting (begin_events) respectively ending (end_events). Sets are not available in Promela as a built in data type, so they are modeled as channels (the Promela model contains a macro definition of the form: '`#define set chan`'). The external_events parameter is a reference (pointer) to the set of actual events that happen, to be updated by the context. The process itself can add events to this set when starting and when ending such that other processes can be made aware thereof. The events to add are the last two parameters of the process sig_start and sig_end. Whether these events should be added or not really depends on the context, whether some other process needs to know. If no process needs to know the parameter is negative, and it will not be added.

The last parameter (depth) to the process is a model-parameter, introduced by the user in the drilling activity (line 18). Recall that any local "variable" of an activity in Aml can be a parameter in case an instantiating activity passes a value to this variable. The drilling activity has 3 local variables: hole, depth, and power_use, but only the first two of these are real parameters instantiated at call time in the experiment activity:

```
  drill with ("hole1" -> hole, 7 -> depth)
```

However, only the depth parameter influences planning since it impacts how much power is used (lines 20 and 25). Note that Map performs a data flow analysis of the Aml model in order to determine which variables are used in planning. The string variable "hole" does not influence the planning, and hence is abstracted away.

The body of the drilling process is divided into two atomic statements, representing respectively the beginning and the end of the activity. The explanations of the two blocks are similar. The beginning block starts by waiting for the events in the begin_events set to become subset of the external_events set (subset(begin_events,external_events)). The various operations on sets are really operations on channels, modeling set addition, set membership test, and subset test. It then performs checks on and assignments to various resource, state and semaphore variables. A conditional statement `condition -> statement` causes the process to block until the condition becomes true (Promela semantics). Finally, it is signaled to the external_events set that the process has started (if the sigstart value is not negative). The addorlog(set,signal) function adds the signal to the set, if the signal is not negative, and furthermore stores the signal in a global variable `_event_` (such that LTL formulas can refer to it) of an enumerated type of all the possible events, one for the beginning and end for each activity.

```
mtype {
  BEGIN_drill, END_drill,
  BEGIN_uplink, END_uplink,
  BEGIN_charge, END_charge,
  BEGIN_experiment, END_experiment
}
local mtype _event_;
```

The experiment activity is in a similar fashion translated into the Promela process shown in Figure 7. This process declares two variables. The set-variable events will be updated continuously during execution and will contain the events that occur during an experiment (third external_events parameter to the sub-activities). The set-variable end_drill is initialized once to contain the set of events that the drill activity has to wait for before it can end. These required sizes of these sets (3 and 1) are calculated at translation time. For example, 3 events will need to be recorded: end of charge (needed by the drill), and end of drill and uplink (needed by the experiment that cannot terminate before these have terminated, see Figure 5.

The body of the first atomic block is a conditional `if :: ... :: ... fi` statement, having two entries that are chosen non-deterministically, corresponding to the `or` operator occurring in line 42 of the AML model in Figure 2. The form of the two choices are similar. In the first case, corresponding to lines 40-41 of Figure 2, the set end_drill is created to contain the event END_charge by the statement: mustwaitfor(end_drill,END_charge), which adds its second argument to the first argument set. This set is then passed as the second argument to the drill activity in the subsequent line to indicate that the drill activity has to wait for the charge to end before it can end itself. The other event sets passed around are empty (nullset). The events passed as arguments, for example the negative -BEGIN_charge and the positive END_charge to the charge activity, indicate that no-one cares about when a charge begins (negative so it will not be added to the events set), whereas for example the drill activity needs to know when the charge ends. Finally, the experiment will not continue before the sub-activities spawned in each branch have terminated (isin(...)).

The AML model contains in line 20, Figure 2, a call of the function powerof, which must have been defined as a C++ function in a separate file. MAP does not translate these C++ functions. Instead, their occurrence in the AML model is marked by the translator, and a user has to program these as PROMELA macros, as in this case: `#define powerof(depth) (depth*10)`.

5 The Earth Orbiter 1 Application

ASPEN has successfully commanded (and is still at the time of writing commanding) the Earth Observer 1 (EO1) Autonomous Sciencecraft Experiment onboard the EO1 earth orbiting satellite. The EO1 satellite orbits earth, taking photos of the surface and comparing recent images with previous images to detect changes due to, for instance, flooding, fire and other natural events. Upon detecting a change, the spacecraft soft-

```
proctype experiment(set begin_events; set end_events;
                set external_events; short sigstart; short sigend)
{
  byte _e_;
  set events = [3] of {mtype};
  set end_drill = [1] of {mtype};
  atomic {
    subset(begin_events,external_events);
    addorlog(external_events,sigstart);
    if
    ::
      mustwaitfor(end_drill,END_charge);
      run drill(nullset,end_drill,events,-BEGIN_drill,END_drill,7);
      run uplink(nullset,nullset,events,-BEGIN_uplink,END_uplink);
      run charge(nullset,nullset,events,-BEGIN_charge,END_charge);
      isin(END_drill,events) && isin(END_uplink,events) &&
          isin(END_charge,events)
    ::
      run charge(nullset,nullset,events,-BEGIN_charge,END_charge);
      isin(END_charge,events)
    fi
  };
  atomic {
    subset(end_events,external_events);
    addorlog(external_events,sigend)
  }
}
```

Fig. 7. Promela model of experiment activity.

ware generates a new goal to take a more detailed follow-up image and Aspen generates a plan to achieve that goal.

Our original goal was to enable Map to convert the EO1 Aml model into Promela. The EO1 model features the most commonly used Aml constructs, and therefore, a tool that can convert this model will be capable of converting a very broad set of realistic Aml models, a non-trivial achievement. With well over 100 activities in the EO1 Aml model, and an ever changing set of goals, EO1 also illustrates that an automated conversion tool is necessary to make the logic model checking of APS input models practical.

EO1 has two imaging instruments that can read from and write to a solid state recorder. The designers of the Aml model were concerned about a possible data race on the state recorder, violating that reads and writes must mutually exclude each other. This property was formulated in Promela using a counter and was shown not to be violated on a very large state space, although not the complete state space. The model analyzed is approximately 7300 lines of code, causing approximately 4000 lines of Promela code to be generated. Two experiments were performed, each applying Spins bit-state hashing where not all of the state space is explored. Each experiment was performed comparing single core (1 CPU) and multi-core (8 CPUs) runs. In the first experiment 10 million states were explored using 11.6 minutes on 1 CPU and 89 seconds on 8 CPUs. In the second experiment, with more aggressive bit-state hashing, 2.5 billion states were explored, using 2.6 days on 1 CPU and 8 hours on 8 CPUs.

6 Conclusion and Future Work

The translator translates a large subset of Aml relatively faithfully by attempting to map Aml source constructs to Promela target constructs, which are supposed to yield a behavior in Spin similar to the behavior of the source in Aspen. However, some parts of Aml are not translated, in some cases as an optimization mechanism. The main constructs of Aml that are not translated include time values and durations, reals and floats, priorities, and a special form of call-by-reference parameter passing that Spin does not support. Hence, an activity cannot influence the value of the actual parameter. Of these omitted concepts, some are generally hard to translate, such as time, real numbers, and call-by-reference of activities. The remaining omissions could be handled more easily. The Map tool shall be seen as an aid in examining the utility of model checking in testing plan models. Future work includes examining exactly what forms of verification can be performed with the presented tool that cannot easily be performed with Aspen.

References

1. M. Smith, G. Holzmann, G. Cucullu, B. Smith, "Model Checking Autonomous Planners: Even the Best Laid Plans Must be Verified"', IEEE Aerospace Conference, Big Sky, Montana, March, 2005
2. Gerard Holzmann, The Model Checker Spin, IEEE Transactions on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279-295.
3. Gerard Holzmann, The Spin Model Checker: Primer and Reference Manual, 2003, Addison-Wesley, ISBN 0- 321-22862-6, 608 pgs.
4. http://www.spinroot.com
5. L. Khatib, N. Muscettola, K. Havelund Verification of Plan Models using UPPAAL, First Goddard Workshop on Formal Approaches to Agent-Based Systems. NASAs Goddard Space Center, Maryland. March 2000.

6. K. Havelund, M. Lowry, J. Penix, Formal Analysis of a Space Craft Controller using Spin, IEEE Transactions on Software Engineering, Vol. 27, No. 8, August, 2001. Originally appeared in proceedings of the 4th SPIN workshop, Paris, France. November 1999.
7. K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, J. L. White, Formal Analysis of the Remote Agent - Before and After Flight, The Fifth NASA Langley Formal Methods Workshop, Virginia. June 2000.
8. M. Feather, B. Smith, "Automatic Generation of Test Oracles: From Pilot Studies to Applications," Proceedings of the Fourteenth IEEE International Conference on Automated Software Engineering (ASE-99), Cocoa Beach, FL. October, 1999. IEEE Computer Society, pp 63-72.
9. J. Penix, C. Pecheur, K. Havelund, Using Model Checking to Validate AI Planner Domain Models, 23 Annual NASA Goddard Software Engineering Workshop, Goddard, Maryland, Dec 1998.
10. B. Cichy, S. Chien, S. Schaffer, D. Tran, G. Rabideau, R. Sherwood, Validating the Autonomous EO-1 Science Agent, International Workshop on Planning and Scheduling for Space (IWPSS 2004). Darmstadt, Germany. June 2004.
11. M. Smith, G. Holzmann and K. Ettessami, Events and Constraints: a Graphical Editor for Capturing Logic Properties of Programs, 5th International Symposium on Requirements Engineering, pp 14-22, Toronto, Canada. August 2001.
12. S. Chien, R. Knight, A. Stechert, R. Sherwood, G. Rabideau, Using Iterative Repair to Improve Responsiveness of Planning and Scheduling, International Conference on Artificial Intelligence Planning Systems (AIPS 2000). Breckenridge, CO. April 2000.
13. Alex Fukunaga, Gregg Rabideau, Steve Chien, ASPEN: An Application Framework for Automated Planning and Scheduling of Spacecraft Control and Operations, Proceedings of International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS97), Tokyo, Japan, 1997, pp. 181-187.
14. B. Smith, R. Sherwood, A. Govindjee, D. Yan, G. Rabideau, S. Chien, A. Fukunaga, Representing Spacecraft Mission Planning Knowledge in Aspen, AIPS-98 Workshop on Knowledge Engineering and Acquisition for Planning, June 1998. Workshop notes published as AAI Technical Report WS-98-03.